TU Clausthal

Clausthal University of Technology

LightJason: A BDI Framework Inspired by Jason

Malte Aschermann, Philipp Kraus, Jörg P. Müller

Ifi Technical Report Series **Ifi-16-04**

Department of Informatics
Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Tobias Ahlbrecht
Contact: tobias.ahlbrecht@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

LightJason: A BDI Framework Inspired by Jason

Malte Aschermann, Philipp Kraus, Jörg P. Müller

Department of Informatics, Clausthal University of Technology, Julius-Albert-Str. 4
D-38678 Clausthal-Zellerfeld, Germany
{malte, philipp}@lightjason.org, joerg.mueller@tu-clausthal.de
http://lightjason.org

Abstract

Current BDI agent frameworks often lack necessary modularity, scalability and are hard to integrate with non-agent applications. This paper reports ongoing research on *LightJason*, a multi-agent BDI framework based on *AgentSpeak(L)*, fine-tuned to concurrent plan execution in a distributed framework; *LightJason* aims at efficient and scalable integration with existing platforms. We state requirements for BDI agent languages and corresponding runtime systems, and present the key concepts and initial implementation of *LightJason* in the light of these requirements. Based on a set of requirements derived for scalable, modular BDI frameworks, the core contribution of this paper is the definition of a formal modular grammar for *AgentSpeak(L++)*, a modular extension of *AgentSpeak(L)*, and its underlying scalable runtime system. A preliminary validation of *LightJason* is given by means of an example evacuation scenario, an experimental analysis of the runtime performance, and a qualitative comparison with the Jason platform.

1 Introduction

Agent-oriented programming (AgOP) [1] is about building systems consisting of *software agents* maintaining mental states, based on declarative logical languages. The Belief-Desire-Intention (BDI) paradigm [2] has become the prevalent approach to AgOP and multi-agent systems (MAS). Such agent programs consist of statements in first-order logic, allowing agents to deduce new facts, commit to plans and eventually execute actions. A very popular language for programming BDI agents is *AgentSpeak(L)* [3]. *Jason* [4] has been instrumental to the popularity of *AgentSpeak(L)* by providing a BDI agent framework that combines an extension of *AgentSpeak(L)* with an interpreter and provides integrated development environment (IDE) plugins for JEdit and Eclipse.

Table 1: The ten most commonly used programming languages in 2016. C, C++, C#, Java, Python, Ruby represent the mainly used functional and object-oriented languages in world-wide projects (percentage shows the spread within projects).

Rank	Tiobe		PopularitY		RedMonk
1	Java	(20.794%)	Java	(22.4%)	JavaScript
2	C	(12.376%)	Python	(14.9%)	Java
3	C++	(6.199%)	C#	(10.7%)	PHP
4	Python	(3.900%)	C++	(10.2%)	Python
5	C#	(3.786%)	PHP	(9.4%)	C#
6	PHP	(3.227%)	Javascript	(7.1%)	C++
7	JavaScript	(2.583%)	C	(5.9%)	Ruby
8	Perl	(2.395%)	Objective-C	(3.8%)	CSS
9	Visual Basic .NET	(2.353%)	Matlab	(3.1%)	C
10	Ruby	(2.336%)	R	(2.5%)	Objective-C
...	...	...	...	...	...
33	Prolog	(0.471%)	—		—

However, analysing the level of usage of BDI agent frameworks in software engineering practice reveals a sobering picture. A look at the major programming indices Tiobe [5], Redmonk [6] and PopularitY [7], which measure the popularity of programming languages, shows that the world of practice is still dominated by imperative and object-oriented languages (see Table 1). Only Tiobe lists any logic-based languages: The major proponent *Prolog* is ranked 33rd. AgOP languages are not represented at all.

Furthermore, in their study of MAS application impact, [8] show that among the agent languages, the only 'true' BDI language with some application impact is Jack, a proprietary language, while the use of languages like *Jason* or *GOAL* is restricted to academic prototypes.

The hypothesis underlying our research is that part of the reasons[1] for this dire state are elementary shortcomings of AgOP languages regarding modularity, maintainability, software architecture interoperability, performance, and scalability.

This paper reports ongoing research on a multi-agent framework based on *AgentSpeak(L)* which aims at an efficient and scalable integration into existing platforms, enabling non-agent-aware systems to incorporate agent-based optimisation techniques to solve distributed problems. We present the initial version of *LightJason*, a BDI agent framework fine-tuned to concurrent plan execution in a distributed environment. *LightJason* provides a runtime system for *AgentSpeak(L++)*, an extension of *AgentSpeak(L)*. The

[1] We acknowledge that there are various other reasons including the difficulty of maintaining long-lived software projects in an academic context.

runtime system is a Java-based instantiation of the *Multi-Agent Scalable Runtime platform for Simulation* (*MASeRaTi*) architecture presented in [9].

In Section 2, we state requirements for an AgOP framework and provide a brief analysis of related work. Section 3 sketches the overall design approach and architecture of *LightJason*. Section 4 is the core of this paper. It describes the modular design and grammar of the *AgentSpeak(L++)* language, including the built-in action concept. Section 5 presents a preliminar validation based on an example application and a performance validation. The paper ends with a conclusion and outlook in Section 6.

2 Requirements and State of the Art

2.1 Requirements

Over the past years, we have gained experience in modelling and engineering multi-agent applications based on the BDI paradigm (most notably in domains of traffic and industrial business processes), but also with developing agent programming languages and runtime platforms. While we consider the BDI abstraction appealing and intuitive for modelling sociotechnical systems, we were often confronted with the limitations of today's agent platforms. From these limitations, we derived a number of requirements for BDI agent platforms, which extend the list of general requirements from [4, p. 7]):

1. Integration in existing software architectures: E.g., for agent-based traffic simulation, MAS framework need to scale up for thousands of agents, while connecting to an existing runtime system (e.g., a SUMO traffic simulator). Thus, agent behaviour needs to be attached on top of an existing object-oriented structure. One desirable (clean and efficient) way of doing this is by subclassing a generic agent class to implement domain-specific functions.

2. Modularisation of agents and underlying data structure: There is a lack of modular logic languages which are suitable for extending existing software frameworks. Modularity is a key requirement also to enable scalable integration with legacy systems. It is also key for maintainability of software.

3. Agent scripting language with strict language syntax: This will allow us to provide tools helping the developer to check agent program syntax at parse-time, i.e., before the agent is executed. Also, the language should reflect modularisation (see above).

4. Action checking during parsing time, not during run time: Methods of an agent-object should create actions automatically and availability of

actions in the Environment should be checked on parsing time. The absence of such a mechanism in *Jason* is a source of errors and tedious debugging.

5. Full execution control of the reasoning cycle: The fixed BDI agent reasoning cycle as e.g., implemented in *Jason* leads to problems in conjunction with application-level processes that require synchronisation among agents. An example reported in [9] is the Nagel-Schreckenberg car following model. Thus, it is desirable to generalise the *action-centric* reasoning cycle in *Jason* to a *plan-centric* model, that (1) allows the agent class designer to modify the agent cycle for application-specific agent implementations, and (2) provides ways for scenario-specific modifications of the implementation of the execution mechanism (thread pool).

6. Parallel execution of plans in separated execution tasks: Related to the previous requirement, a AgOP framework should enable concurrent execution of agents (and thus plans), so that in principle, on every cycle the plans for all possible goals that can be instantiated will be executed; hence, when the cycle ends, all plans are finished.

7. Agent generation mechanism to allow easy creation of large numbers of agent instances: For generating new agents, a Factory pattern should be provided, with only a single parsing process for better modularisation.

8. Hierarchically structured belief bases and actions in semantic groups: Hierarchical naming schemes are useful for modularity and efficient organisation of different knowledge categories.

2.2 Discussion of state of the art

The main concepts of BDI frameworks are mostly based on the Procedural Reasoning System (PRS) [10, 11] and the first robust implementations such as dMARS [12]. As [13] and subsequent surveys point out, virtually all existing multi-agent frameworks are not designed for productive use (performance, scalability) and easy integration with specific domains. The design of agent-based scripting languages leads to challenges in maintainability; e.g. Bordini et al. [14, p. 1300] state that: *"[T]he AgentSpeak(L) code is not elegant at all. The resulting code is extremely clumsy because of the use of many belief addition, deletion, and checking (for controlling intention selection) [...] [and] thus a type of code that is very difficult to implement and maintain."* Though this is a paper from 2002, the situation has not changed much.

MAS platforms like *Jason* provide a separate runtime system, these approaches raise issues regarding scalability and consistency, especially when

combining existing systems with MAS. In the case of *Jason*, this also can lead to ill-defined execution behaviour of agents, especially regarding clarity when an iteration of the agent control cycle has ended (see Item 5 above).

In this paper, we focus on the comparison with *AgentSpeak(L)/Jason* as the most prominent (open-source) representative of BDI languages / platform. We compared the legacy *Jason* 1.4 branch, which is still in use in our research group for small-scale agent-based traffic simulation (e.g. [15]), and the quite recently published *Jason* 2.0 branch with our requirements. *Jason* 1.4 lacks support for all the above-mentioned requirements except a partially support for modular agents (requirement 2), due to its `include` functionality. *Jason* 2.0 additionally supports a hierarchical structuring of agents (requirement 8), but this feature is limited to beliefs and plans[2]. Also, one new feature of *Jason* 2.0 is parallel execution of plans [16], which addresses requirement 6. However, like *Jason* 1.4, *Jason* 2.0 still heavily relies on synchronised data structures in their architecture design, implying slow-downs due to locking and CPU context switches during each agent cycle. In their approach adding concurrency to the reasoning cycles in *Jason*, [16] provided benchmark results regarding scalability; their test setup with only two CPU cores and synthetic benchmarks (e.g. nested for-loops and Fibonacci sequence) resulted in a linear increase in execution time for up to 500 agents, which would also be expected for single-thread applications.

In order to tackle the above requirements, we start from the architecture design of *MASeRaTi* [9], as an attempt to tackle the scalability issues in modern MAS. In the next sections, we describe how we take this approach forward. We create a modified, light version of *AgentSpeak(L)* (named *AgentSpeak(L++)*) and build a Java-based implementation of the *MASeRaTi* architecture.

3 *LightJason* Architecture and Data Model

In this section we describe the methodology underlying the design and the implementation of *LightJason*, aiming at scalability, concurrent execution, and modularity.

There is broad agreement in the AgOP literature that *"[a] multi-agent system is inherently multithreaded, in that each agent is assumed to have at least one thread of control [17, p. 30]"* meaning that agents should be able to pursue more than one objective at the same time. To implement this conceptual notion of concurrency at the technical level, we refer to the basic notion of a thread [18] as a *"lightweight process"*, and that all threads are running within the same process. Thus, in *LightJason*, an agent is be controlled by a thread

[2] `https://github.com/jason-lang/jason/blob/master/doc/tech/modules-namespaces.pdf`

during the reasoning process and stores all data for the reasoning internally, by following the thread-local-storage model. A multi-agent system runs on a thread-pool in which agents are running in an asynchronously and continuous manner. For each agent, actions will be executed independently and immediately within the environment.

Our general approach in *LightJason* is to conceive *AgOP* as a combination of *Imperative*, *Object-Oriented* and *Logic Programming*, see Figure 1.

We see *AgOP* as a paradigm which extends the Object-Oriented Programming (OOP) paradigm (see Figure 1), but in our view this extension is more than just adding new syntax features to the language.

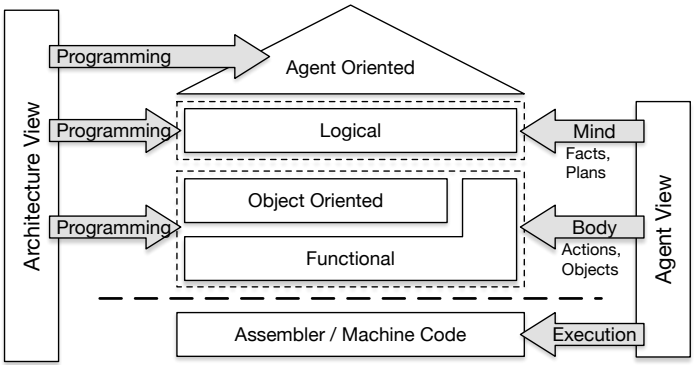


Figure 1: Agent Views at Design and Implementation Time

Using declarative *Logic Programming* models, we define flexible agent behaviour. Mostly, an agent should cooperate with other agents in a decentralised way with local views to the problem. These components are described within the BDI architecture (belief, desires and intentions). To get into a more detailed view, an agent is not one single software component but it is split up into two different elements, i.e. *agent-mind and agent-body*. This approach is a reverence to the Mind-Head-Body model proposed by Steiner et al. in the *Multi-agent Environment for Constructing Cooperative Applications (MECCA)* architecture [19].

Considering the *semantic* and *syntactic* view on MECCA we generalised its architecture and reduced the authors' concepts of cooperation in order to achieve similar complexity by using current technologies and modelling concepts. From the semantic point of view MECCA's definitions and algorithms are mapped to (logical) algorithms or complex components and from the syntactic point of view MECCA's elements are mapped or generalised to logical elements of *LightJason*. MECCA uses cooperative primitives, which can be plans, goals or tasks ([19, Chapter 3]). As such primitives are a central part of the *agent-mind*, we adopted them directly into our *AgentSpeak(L++)*

definition and syntactically denoted plans and goals as *literals*. We inherited MECCA's tasks as actions, which we also represented as *literals*. But in contrast to our work MECCA uses a more detailed structure, e.g. for scheduled plans [19, Chapter 3.1], which are based on a cost function. Our concept does not support such scheduled plans from an explicit modelling perspective, but supports the cost function concept and plan-scheduling during runtime.

LightJason uses a more generalised and parametrisable approach, but is conceptually very similar to MECCA. In each cycle *LightJason*'s agents execute every plan that can be instantiated, i.e. plans which *plan conditions* evaluate `true`: This instantiation process is called the *unification process* of a goal literal into a matching plan literal. After that the plan condition, which is also supported in MECCA [19, Table 1], will be evaluated. This plan condition can contain a *score value* (representing the cost for each plan's execution), which can be set by the underlying software component.

Another concept we adopted from MECCA is the *head-communicator-interface* [19, Chapter 3] which allowed us to describe the agent as a "data-streaming process". From the software-architecture point of view *agent-mind* and *agent-body* are detached software components. The *agent-mind* is a generic component, which gets implemented in the *AgentSpeak(L++)* language, whereas the *agent-body* is designed as a domain-dependent component. In our approach the *body* creates the logical elements and calls the *mind's* functions with literals as parameters. We implemented this mechanism by using a streaming process, in which data can flow between *body* and *mind* continuously to maximise efficiency and throughput. This approach allowed us to incorporate MECCA's communication structure by using streams as means of communication between software components. We generalised this concepts more from the technical perspective: Plans, beliefs, actions in *AgentSpeak(L++)* are implemented by OOP components in Java. Each Java component inherits from *Object class* and from a technical standpoint each variable is a pointer to an object. *LightJason* translates the *AgentSpeak(L++)* code into Java objects and all agents, which base on the same *AgentSpeak(L++)* source, share the same plan objects, because all objects are referenced by pointers.

Based on the two-layer agent structure we implemented *LightJason* in an Object-Oriented language (Java 8) with additional features of logic programming languages and imperative components to describe the execution plan sequence (see Figure 1).

The symbolic representation of an agent's mind is stored as *logic literals*, as in *Prolog* or *AgentSpeak(L)*. All literals of an agent are stored within a belief base for getting access during runtime. During the agent's execution the agent asks for particular literals. This is realised by *unification*. As this process is run many times, we optimised the internal data structure representing the logic elements for parallel execution and avoiding cost-intensive back-

tracking.

The *Imperative Programming* paradigm is used to describe the execution behaviour of agents in *LightJason* (similar to the Patterns of Behaviour (PoBs) in the INTERRAP architecture [20]). In contrast, to INTERRAP, we provide for parallel execution of PoBs, so that each actions, assignment or expression can be run or evaluated in parallel.

Finally, *LightJason* is Java-based; the internal representation of agents is written in an Object-Oriented style. With Java version 1.8, also functional programming with lambda-expressions[3] and streams[4] are supported, which we make extensive use of, to further parallelise execution and gain more scalability. By relying on a structured OOP design with *concurrent data structures* we can create inheritable agent objects running in a multithreading context, allowing easier integration with domain-specific software systems.

4 *AgentSpeak(L++)* Language Definition

We regard an agent as a hybrid system, which combines different programming language paradigms, allowing programmers to describe complex behaviour. This abstract point of view allows a flexible structure – also for non-computer scientists – to parameterise or specify a software system. On the one hand, we needed a syntax definition for defining behaviour, on the other hand, we need a straightforward and clean syntax. The whole syntax was designed as a *logic programming language*, by which all elements could be reduced to *terms*[5] and *literals*[6], which define a symbolic representation of behaviour and (environment) data. This allows modelling a generalised multi-agent system, which can later be concretised for different applications, i.e. scenarios and supports the agent programmer to design the behaviour by scripting beliefs, rules, plans and actions. The multi-agent system then "translates" the data between the Object-Oriented back-end to a symbolic structure in the front-end for modelling behaviour, which can be seen as a *wrapper around datasets*.

4.1 Grammar Definition

Our first contribution is the definition of a *scripting language* based on a modified and extended *AgentSpeak(L)* grammar. For building the *lexer* and *parser*

[3] `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`

[4] `https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`

[5] Term: `https://git.io/viKWQ`, EBNF: `https://git.io/viKWx`

[6] Literal: `https://git.io/viKlt`, EBNF: `https://git.io/viKlI`

components, we used the Java AntLR[7] framework. The framework generates all required software components based on the grammar definition. The result of this generation process is a tree data structure, which can be processed by an abstract syntax tree (AST) visitor.

We modularised the grammar into subgrammars to obtain a more abstract structure of the agent programming language (Figure 2). This allowed us to get a more flexible parsing component, which could be split up into a layer-based structure. In designing the language, we paid utmost attention to create a clean and human-readable source code for the LightJason platform.

Figure 2 shows the main structure of our grammar definition. We referenced the implementation in Backus-Naur form (for AntLR usage) and visualisation with a railroad diagram (RRD) in the repository. A full listing of the grammar can be found in appendix A.

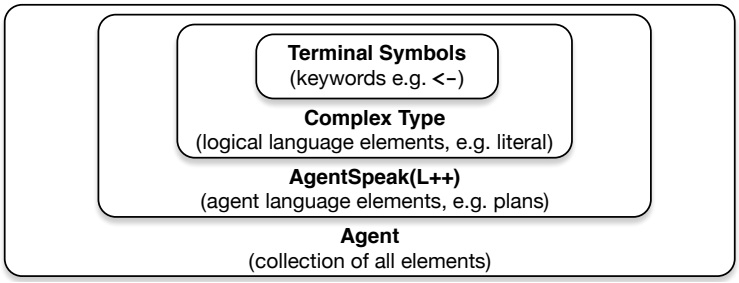


Figure 2: Modular Grammar Structure

On the language definition the ordering of the plans in each agent within the source-code file is negligible for the runtime process. In the agent cycle all plans, for which the plan condition evaluates *true*, will be triggered in parallel. As can be seen in the following grammar modules, *belief*, *plan* and *actions* can be grouped into hierarchical naming structures by slashes (/) or minuses (-). For the *plan-bodies* we used a more functional definition of the elements (based on C/C++, Java syntax), therefore elements are semicolon-separated.

4.1.1 Grammar module: Agent

The agent is structured and modularised as described in Section 3 and Figure 2. We defined the *agent grammar*[8] to represent the top-level structure of an agent. The underlying grammar structure, as can be seen in Section 4.1.3, is included into the agent's grammar. Based on on the structure in Figure 2

[7] `http://www.antlr.org/`
[8] Grammar Agent: `https://git.io/viKsp`, EBNF: `https://git.io/viKsj`

the agent's grammar can be seen in the RRD of Figure 3. For the full grammar specification refer to appendix A.1.

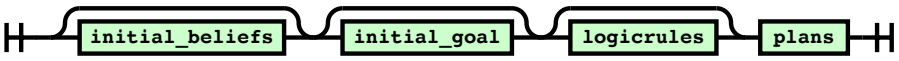


Figure 3: Agent RRD

4.1.2 Grammar module: Plan Bundles

For helping programmers implementing sophisticated behaviour of agents in complex scenarios, we added *plan bundles* as additional structures. These *collections of different plans, logical rules and beliefs* are depicted in Figure 4 (a formal definition can be seen in appendix A.2). The bundle structure can be used for defining some generic models of agents and can be read into the agent during instantiation. With this definition the agent's behaviour can be modularised and structured. A plan bundle is not directly instantiable and can be used only in combination with an agent.

Figure 4: Plan bundle RRD

4.1.3 Grammar module: AgentSpeak(L++)

The *AgentSpeak(L++)* grammar definition[9] describes the language parts for implementing the agent's logic and is designed as an *is-part-of* relation to the *agent grammar*. The full grammar structure is referenced in appendix A.3. In addition to the original *AgentSpeak(L)* grammar we added new language structures e.g.

Lambda-Expressions[10] replacing classical loop structures. The expression can be run in sequential and parallel execution mode. The RRD of the generic lambda structure can be shown in Figure 5 The following example shows the baseline definition: *For each element in L, push the element into the variable Y and do something*, e.g.

```
L = collection/list/range(1, 20);
(L) -> Y : generic/print(Y);
```

[9] AgentSpeak(L++): `https://git.io/viKG9`, EBNF: `https://git.io/viKGn`
[10] Lambda-Expression: `https://git.io/viKG2`, EBNF: `https://git.io/viKG0`

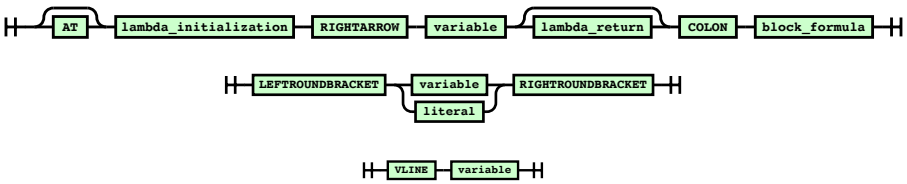


Figure 5: Lambda Expression RRD with initialization and return

Multi-Variable-Assignments[11] replacing Prolog's head-tail notation of lists into a more useful structure. For example the following code will put each element of `L` into the variables `A...G`, with `G` containing the tail of the list. Elements of no further use can be discarded by the anonymous variable "_". The Figure 6 shows the generic structure and the following example the assignment mechanism.

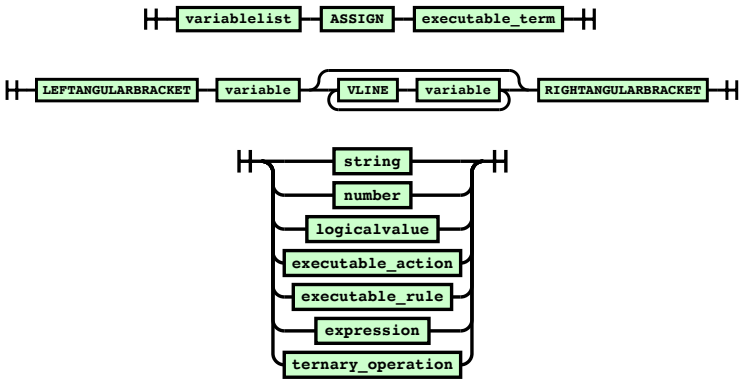


Figure 6: Multi-Variable-Assignments RRD with variable and executable term

```
L = collection/list/range(1, 20);
[A|B|C| _ |D|E|F|G] = L;
```

Explicit repair-formula notation[12] to get a readable structure of *execution chains* for extensive error handling. The general structure in Figure 7 contains a reflexive structure of the grammar rule. The following example shows the use of the grammar structure. If the execution of *plannotexist* fails, the agent tries to run *otherplan* immediately and if this also fails the repair-formula returns *true*, preventing the currently running plan from failing.

[11] Multi-Assignment: `https://git.io/viKZ7`, EBNF: `https://git.io/viKZM`
[12] Repair-Formula: `https://git.io/viKZN`, EBNF: `https://git.io/viKZp`

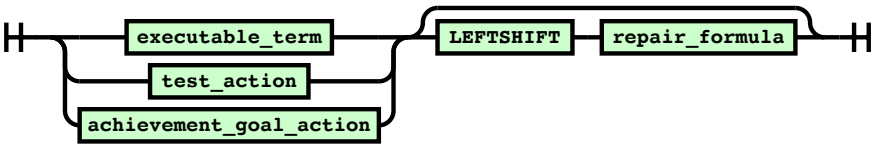


Figure 7: Repair-Formula RRD

```
?plannotexist << !!otherplan << true;
```

Multi-Plan[13] like class structures for a more readable source-code without duplicating plan names. A multi-plan (see Figure 8) is a collection of all plans, which uses equal trigger literals. The plan `main` is defined in

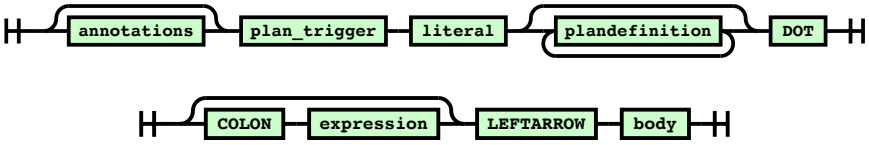


Figure 8: Plan RRD

three parts. The first plan (line 2) is run, iff there exists a belief, which value can be unified to a string, the second plan (line 3) is run iff the belief value can be unified to a number value and the number is greater than 1000 and the third plan (line 4) is run if `main` is triggered (default behaviour of the plan)

```
+!main
   : >>( hallo(X), generic/type/isstring(X) ) <- generic/print("1st
        plan: unification variable", X)
   : >>( hallo(X), generic/type/isnumeric(X) && X > 1000 )  <-
       generic/print("2nd plan: unification of", X)
   <- generic/print("default plan").
```

[13] Multi-Plans: `https://git.io/viKn0`, EBNF: `https://git.io/viKnn`

4.1.4 Grammar module: ComplexType

The ComplexType grammar[14] represents the *logic components* of our language. We are reflecting the Prolog syntax and this structure has got the relation *is-part-of* of the *AgentSpeak(L++)* grammar. The full grammar listing is referenced in appendix A.4. The main Prolog structure uses *terms*, *literals*, *atoms* and *variables*. We are using this baseline structures and extend the definition.

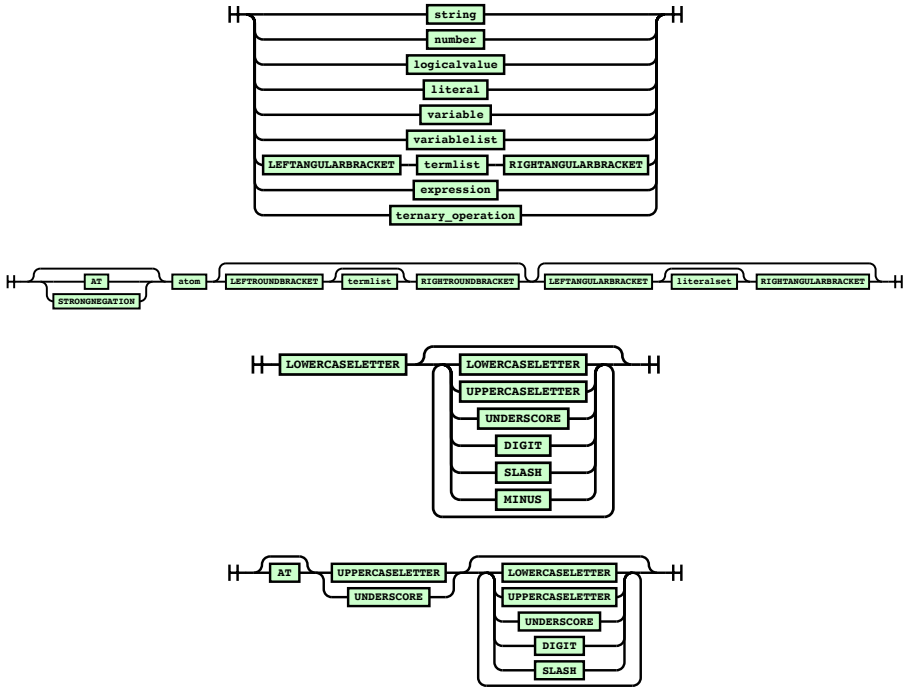


Figure 9: Logical Language RRD with term, literal, atom and variable definition

Built-in numerical constants[15] added for calculations: `avogadro`, `boltzmann`, `electron`, `euler`, `gravity`, `infinity`, `lightspeed`, `neutron`, `pi`, `proton`

Floating- and Integer representation[16] of numbers for different arithmetic execution

[14] ComplexType Grammar: `https://git.io/viKnA`, EBNF: `https://git.io/viKn7`
[15] Numerical Constants: `https://git.io/viKcf`, EBNF: `https://git.io/viKcI`
[16] Number Definition: `https://git.io/viKC8`, EBNF: `https://git.io/viKCE`

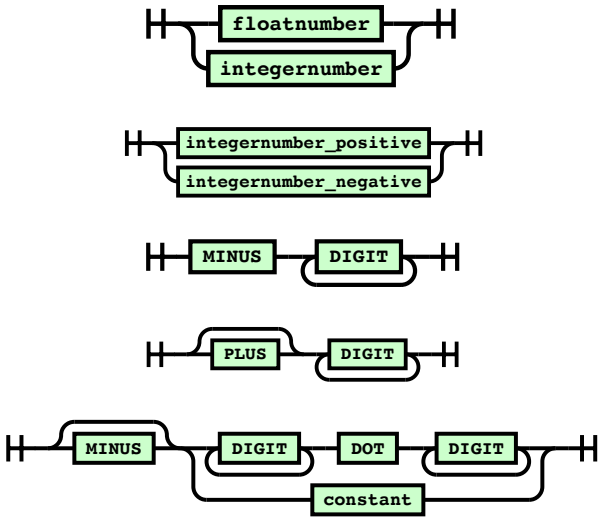


Figure 10: Number structure RRD with integer (positive & negative) and floating point number definition

Parallel execution notation allows to define thread-safe variables, parallel execution of actions, lambda-expression or logic-rule execution

```
L = collection/list/range(1, 20);
@(L) -> Y | R : R = Y+1;
@>>( hallo( Unify ), generic/type/isnumeric( Unify ) &&   (Unify >
    200) )
```

Ternary operator[17] to remove classical if-else statements to get a more readable code

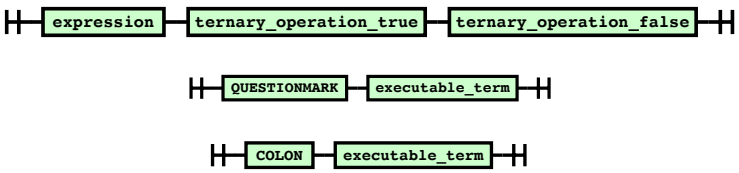


Figure 11: Ternary operator RRD with true and false part

```
Z = math/statistic/randomsimple();
Text = Z > 100 ? "Z greater equal 100" : "Z is less 100";
generic/print("ternary operator", Text);
```

[17] Ternary Operator: `https://git.io/viKCg`, EBNF: `https://git.io/viKCo`

Variable execution[18] to executed variables directly as a plan- or logic-rule definition

```
PLAN = "mytest";
!!PLAN;
!!PLAN(5);
```

4.1.5 Grammar module: Terminal

The terminal symbols[19] of the grammar were defined in a structure, which is in the relation *is-part-of* of the ComplexType grammar. This structures allows fast modification of our grammar. Some features are:

- Strings can be written in single and double quotes,
- numerical constants are defined with human-readable names and
- different styles of source-code documentation are supported.

All terminal symbols are listed in appendix A.5

4.2 Built-in Actions

The language structure and the underlying architecture of our implementation allows to create a flexible action interface. From a software-developing perspective an action is a method inside a class. An agent can only be successfully instantiated if every denoted action exists. To ensure this, every action is checked during parse-time. If the action does not exist, the parsing process fails. In comparison to Jason, we can detect before the agent is running, if the agent source code is syntactically correct and all actions can be executed. The built-in actions are organised in packages. In our framework we currently support actions regarding the following categories: *Bindings* to Java objects, *general* (e.g. print output, agent sleeping or converting data types), *string* operations, *math*, *Basic Linear Algebra Subprograms (BLAS)*, *interpolation* algorithms, *Linear Programming*, *statistics*, *cryptographic* actions, *collections* (e.g. sets, tuples, (multi-)maps and lists), *date and time* manipulation. For a complete overview of these actions and how they can be implemented, we refer to our unit test agent[20] in appendix B.

[18] Variable Execution: `https://git.io/viKWn`, EBNF: `https://git.io/viKWB`
[19] Terminal Symbols: `https://git.io/viKWm`, EBNF: `https://git.io/viKWt`
[20] `https://git.io/vi67u`

Figure 12: Map of built-in actions packages

5 Evaluation and Discussion

5.1 Evacuation scenario

In this chapter, we illustrate the capabilities of *LightJason* by presenting a grid-based evacuation scenario, where agents need to reach an exit to leave the grid. We chose a scenario with 250×250 cells and rectangular obstacles as depicted in Figure 13 and animated in Figure 14. Each agent received the

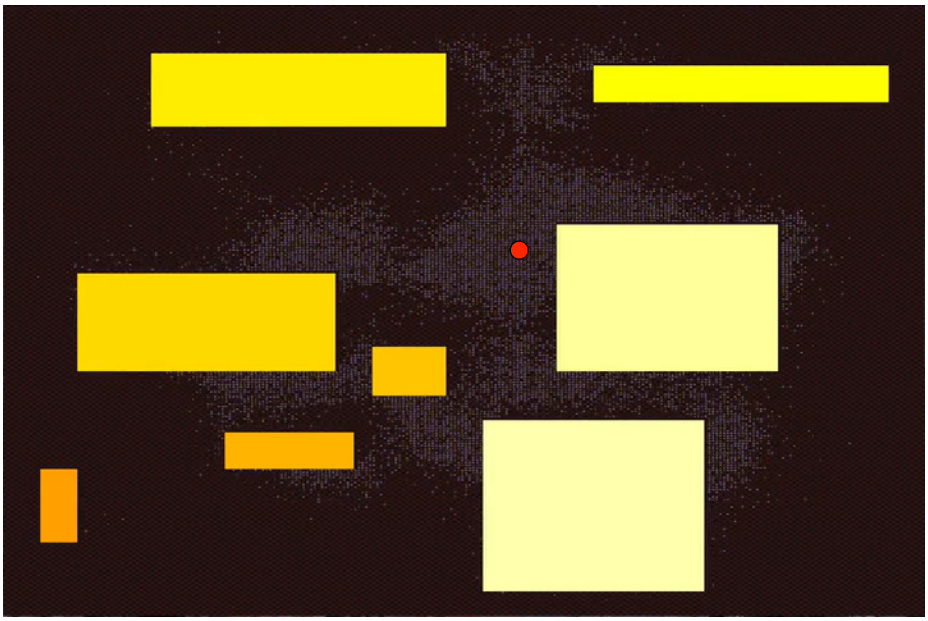


Figure 13: Screenshot of evacuation scenario. Obstacles, agents need to bypass, are shown as yellow rectangles, the exit as a red dot.

same exit destination $(140, 140)$; it disappeared once it reached the approximate destination (± 10 cells). The *AgentSpeak(L++)* code for the corresponding agent is displayed in Listing 1 below.

For finding a route to the exit the agents used the *Jump Point Search (JPS+) with Goal Bounding* [21] algorithm, which, after an initial $O(n^2)$ preprocessing of the grid, outperforms A^* by two to three orders of magnitude in speed. In our example we grouped all plans which describe a moving behaviour under "`movement/...`" (lines 11, 16, 20, 25, 29, 35, 43) and similarly, plans related to reaching a certain location under "`position/...`" (line 49).

Figure 14: Animation of evacuation scenario.

Listing 1: Walking Agent

```
// initial-goal
!main.

// initial plan (triggered by the initial-goal)
// calculates the initial route
+!main <-
  route/set/start( 140, 140 );
  !movement/walk/forward.

// walk straight forward into the direction of the goal-position
+!movement/walk/forward <-
  move/forward();
  !movement/walk/forward.

// walk straight forward fails then go left
-!movement/walk/forward <-
  !movement/walk/left.

// walk left - direction 90 degree to the goal position
+!movement/walk/left <-
  move/left();
  !movement/walk/forward.

// walk left fails then go right
-!movement/walk/left <-
  !movement/walk/right.

// walk right - direction 90 degree to the goal position
+!movement/walk/right <-
  move/right();
  !movement/walk/forward.

// walk right fails then sleep and hope everything will be
// fine later, wakeup plan will be triggered after sleeping
-!movement/walk/right <-
  T = math/statistic/randomsimple() * 10 + 1;
  T = generic/type/toint( T );
  T = math/min( 5, T );
  generic/sleep(T).

// if the agent is not walking because speed is
// low the agent increments the current speed
+!movement/standstill <-
  >>attribute/speed(S);
  S = generic/type/toint(S) + 1;
  +attribute/speed( S );
  !movement/walk/forward.

+!position/achieve(P, D) <-
  route/next;
  !movement/walk/forward.

// if the agent woke up the speed is set to 1 and the agent
// starts walking to the next goal-position
```

```
55 +!wakeup <-
56   +attribute/speed( 1 );
57   !movement/walk/forward.
```

Lines 6 to 8 define the plan of the initial goal, which calculates a route for the agent from the initialised position to the goal-position $(140, 140)$. The routing action in line 7 calculates a list of landmarks, which are sub-goals for the agent. The agent then tries to follow-up on each landmark to reach the end-position. Between the landmarks the agent uses the plan structure to calculate the next concrete position. After the routing algorithm finishes, the agent starts walking in the next cycle (line 8).

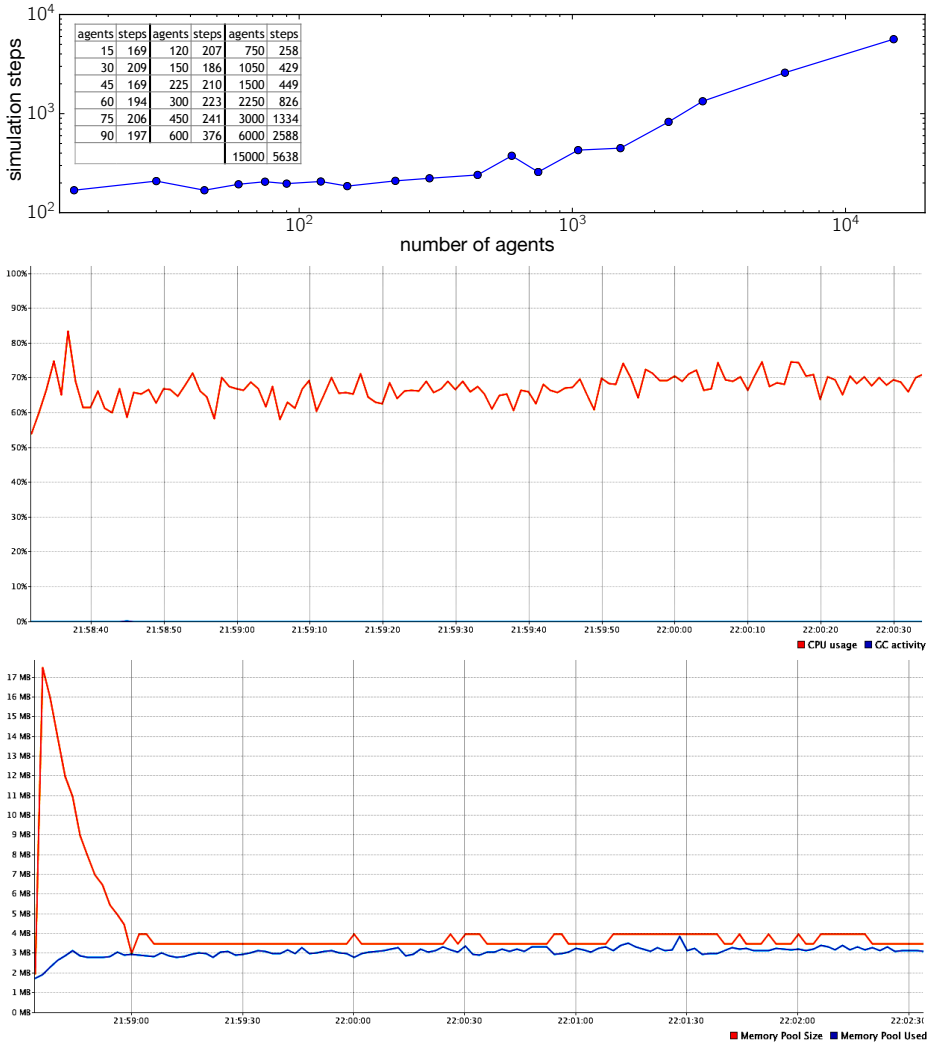
Lines 49 to 51 define the plan, which is triggered by the backend, if the agent is *near-by* (within a radius) the next landmark. The radius is defined similar to the speed belief. In this plan the agent slows down in the vicinity of the landmark.

Lines 55 to 57 define the wake-up plan which is triggered automatically, if the sleeping time ends. The agent resets the current speed to 1 (line 56) and starts walking forward again (line 57).

5.2 Preliminary validation

To validate our results, we conducted first tests with *LightJason* implementation of the evacuation scenario. The goal is to investigate whether the design and implementation of *LightJason* leads to good scalability and cycle consistency regarding the routing model, and number of concurrent running agents. We ran the grid-based scenario as presented in Section 5.1 on an iMac equipped with an Intel® Core™ i7-3770 with 16 GB RAM. Figure 15 illustrates the run-time behaviour of the agents. It (not surprisingly) shows that with an increasing number of agents, each agent needs more cycles to complete its task. This can be attributed to additional invocations of repair plans when an agent's path got obstructed by other agents. This scales sublinearly up to roughly 1000 agents. After that point, the mainly egoistic approach of each agent prevents them to find a free path to the exit, resulting in plan-failures and necessary re-routing.

From a technical perspective we observed that the CPU utilisation is constantly at around 70% for 15000 agents. The constant CPU load shows that the workload induced by the agents is distributed fairly and evenly, avoiding spikes and idle times. Furthermore we observed a low utilisation of the JVM's *survivor space* (roughly 3.5 MB after the initialisation spike), reflecting the design in relying on *lazy bindings* and *LightJason*'s ability to share references to concurrently used data structures, e.g. plans, which only differ in their context and parameters.



agents	steps	agents	steps	agents	steps
15	169	120	207	750	258
30	209	150	186	1050	429
45	169	225	210	1500	449
60	194	300	223	2250	826
75	206	450	241	3000	1334
90	197	600	376	6000	2588
				15000	5638

Figure 15: Results: Number of agents plotted against number of cycles until all agents reached their destinations (top). CPU load (middle) and the JVM's *survivor space* (bottom) for the run with 15000 agents.

5.3 Discussion

In this paper we presented our design and implementation of an agent framework, introducing *LightJason*, an *AgentSpeak(L)* variant. The key aspects we focused on were modularity, flexibility, scalability and deterministic execution behaviour.

The *AgentSpeak(L++)* language supported by *LightJason* reflects *AgentSpeak(L)* as implemented by [4], we differ on a number of aspects, in terms of the language features and – to a larger extent – in the software architecture underlying the implementation. Among others the most notable additions to *AgentSpeak(L)* are lambda-expressions, multi-plan definitions, explicit repair-planning, multi-variable assignments, parallel execution and thread-safe variables.

When considering to port an existing *Jason* code to *LightJason* it is important to understand, that by design in *LightJason* all plans which conditions evaluate to *true* get instantiated. As each agent can instantiate multiple plans concurrently and all agents run concurrently to each other, agents can execute actions which could result in race conditions and may have undesired consequences. Agent developers have to take this into account to avoid unwanted side effects. Here we argue, that in comparison to *Jason*, a non-synchronised system's behaviour results in a considerably more plausible multi-agent system, considering the requirements formulated by [17].

Additional Features Most of the *AgentSpeak(L)* expressions find their equivalents in *LightJason*'s *AgentSpeak(L++)*. Major additions are expressions for parallel execution and unification (`@`). As it is in general possible to design an agent to run plans sequentially, we argue, that for performance reasons it is sensible to make use of parallel execution whenever possible. A second aspect we would like to mention here are plan repairs. *AgentSpeak(L)* recovers from failed plans by making use of the `-!plan` triggering event. In *AgentSpeak(L++)* we extended this concept.

Jason 2.0 With the quite recent release of *Jason* 2.0, there now exist new features[21] in *Jason* which are similar, but independently developed, to some of our own. *Jason* 2.0 introduces *modules and namespaces* to modularise beliefs, goals and plans. In our approach we go even further by integrating those concepts deeply into the fundamental agent grammar. Thus it is possible for us to, for example, modularise actions, functions or beliefs by building hierarchical structures in arbitrary depth allowing greater flexibility than in *Jason*. Another new feature of *Jason* 2.0 are *concurrent courses of actions* [16]. As parallel execution is a fundamental aspect of scalability, we made this an integral part of *LightJason*'s architecture by mainly using Java 1.8 developing techniques and features, e.g *parallel streams* to enable concurrent operations at a very fine granularity.

[21] `https://github.com/jason-lang/jason/blob/master/release-notes.adoc#version-20`

6 Conclusion & Outlook

The contribution of this paper is a flexible agent programming framework *LightJason*, which can be easily integrated into existing systems. The key features of *LightJason* are the simplification of the agent's reasoning cycle and the support of some important requirements including modularity, maintainability, and scalability, combined with state-of-the-art techniques in software development. At the core of *LightJason* is *AgentSpeak(L++)*, a declarative agent scripting language extending *Jason*. We provide a formal grammar definition describing the features of *AgentSpeak(L++)*. For the sake of usability, *LightJason* supports many built-in actions and a structure to load actions in a pre-processing step of the parser. Thus, by parsing the agent's source code it is possible to check that the agent is syntactically correct and can be executed. We further provide generator structures that enable automated creation of large numbers of agents which can be further customised by the user. We also support a fully concurrent and parallel agent execution model of an agent.

This paper describes ongoing work. The next steps will involve a formal definition of the semantics of *AgentSpeak(L++)*. The reader will have noticed that the current *AgentSpeak(L++)* language does not contain language elements for communication. This is intentional, because in our view, communication is a matter of the runtime system rather than of the compilation mechanism. Yet, agent communication is one of the next features to be added to *LightJason*. Also, while we performed an initial qualitative comparison with *Jason*, a thorough experimental benchmarking remains to be performed.

The *LightJason* project can be found under `http://lightjason.org` providing further documentation[22] and source code[23].

[22] `http://lightjason.github.io/AgentSpeak/index.html`
[23] `https://github.com/LightJason/AgentSpeak`

Appendices

A Grammar Definition

A.1 Agent Grammar

agent::= [⟨initial_beliefs⟩? ⟨initial_goal⟩? ⟨logicrules⟩? ⟨plans⟩]

initial_beliefs::= ⟨belief⟩+

initial_goal::= [⟨EXCLAMATIONMARK⟩ ⟨atom⟩ ⟨DOT⟩]

A.2 Plan Bundle Grammar

planbundle::= [⟨belief⟩* ⟨logicrules⟩? ⟨plans⟩]

A.3 *AgentSpeak(L++)* Grammar

achievement_goal_action::= [[⟨EXCLAMATIONMARK⟩
| ⟨DOUBLEEXCLAMATIONMARK⟩] [⟨literal⟩
| ⟨variable_evaluate⟩]]

annotation_atom::= [⟨AT⟩ [⟨ATOMIC⟩
| ⟨PARALLEL⟩]]

annotation_literal::= [⟨AT⟩ ⟨annotation_numeric_literal⟩]

annotation_numeric_literal::= [⟨SCORE⟩ ⟨LEFTROUNDBRACKET⟩ ⟨number⟩ ⟨RIGHTROUNDBRACKET⟩]

annotations::= [⟨annotation_atom⟩
| ⟨annotation_literal⟩]+

assignment_expression::= ⟨assignment_expression_singlevariable⟩
| ⟨assignment_expression_multivariable⟩

assignment_expression_multivariable::= [⟨variablelist⟩ ⟨ASSIGN⟩ ⟨executable_term⟩]

assignment_expression_singlevariable::= [⟨variable⟩ ⟨ASSIGN⟩ ⟨executable_term⟩]

belief::= [⟨literal⟩ ⟨DOT⟩]

belief_action::= [[⟨PLUS⟩
| ⟨MINUS⟩] ⟨literal⟩]

block_formula::= [⟨LEFTCURVEDBRACKET⟩ ⟨body⟩ ⟨RIGHTCURVEDBRACKET⟩]
| ⟨body_formula⟩

body::= [⟨body_formula⟩ [[⟨SEMICOLON⟩ ⟨body_formula⟩]]*]

body_formula::= ⟨repair_formula⟩
| ⟨belief_action⟩
| ⟨deconstruct_expression⟩
| ⟨assignment_expression⟩
| ⟨unary_expression⟩
| ⟨lambda⟩

deconstruct_expression::= [⟨variablelist⟩ ⟨DECONSTRUCT⟩ [⟨literal⟩
| ⟨variable⟩]]

lambda::= [⟨AT⟩? ⟨lambda_initialization⟩ ⟨RIGHTARROW⟩ ⟨variable⟩ ⟨lambda_return⟩?
⟨COLON⟩ ⟨block_formula⟩]

lambda_initialization::= [⟨LEFTROUNDBRACKET⟩ [⟨variable⟩
| ⟨literal⟩] ⟨RIGHTROUNDBRACKET⟩]

lambda_return::= [⟨VLINE⟩ ⟨variable⟩]

logicalruledefinition::= [⟨RULEOPERATOR⟩ ⟨body⟩]

logicrule::= [⟨annotations⟩? ⟨literal⟩ ⟨logicalruledefinition⟩+ ⟨DOT⟩]

logicrules::= ⟨logicrule⟩+

plan::= [⟨annotations⟩? ⟨plan_trigger⟩ ⟨literal⟩ ⟨plandefinition⟩* ⟨DOT⟩]

plan_belief_trigger::= ⟨PLUS⟩
| ⟨MINUS⟩

plan_goal_trigger::= [[⟨PLUS⟩
| ⟨MINUS⟩] ⟨EXCLAMATIONMARK⟩]

plan_trigger::= [⟨plan_belief_trigger⟩
| ⟨plan_goal_trigger⟩]

plandefinition::= [[[⟨COLON⟩ ⟨expression⟩]]? ⟨LEFTARROW⟩ ⟨body⟩]

plans::= ⟨plan⟩+

repair_formula::= [[⟨executable_term⟩
| ⟨test_action⟩
| ⟨achievement_goal_action⟩] [[⟨LEFTSHIFT⟩ ⟨repair_formula⟩]]?]

test_action::= [⟨QUESTIONMARK⟩ ⟨DOLLAR⟩? ⟨atom⟩]

unary_expression::= [⟨variable⟩ ⟨unaryoperator⟩]

A.4 ComplexType Grammar

atom::= [⟨LOWERCASELETTER⟩ [⟨LOWERCASELETTER⟩
| ⟨UPPERCASELETTER⟩
| ⟨UNDERSCORE⟩
| ⟨DIGIT⟩

```
    |   ⟨SLASH⟩
    |   ⟨MINUS⟩ ]* ]
binaryoperator::= ⟨ASSIGNINCREMENT⟩
    |   ⟨ASSIGNDECREMENT⟩
    |   ⟨ASSIGNMULTIPLY⟩
    |   ⟨ASSIGNDIVIDE⟩
constant::= ⟨PI⟩
    |   ⟨EULER⟩
    |   ⟨GRAVITY⟩
    |   ⟨AVOGADRO⟩
    |   ⟨BOLTZMANN⟩
    |   ⟨ELECTRON⟩
    |   ⟨PROTON⟩
    |   ⟨NEUTRON⟩
    |   ⟨LIGHTSPEED⟩
    |   ⟨INFINITY⟩
executable_action::= ⟨literal⟩
executable_rule::= [ ⟨DOLLAR⟩ [ ⟨literal⟩
    |   ⟨variable_evaluate⟩ ] ]
executable_term::= ⟨string⟩
    |   ⟨number⟩
    |   ⟨logicalvalue⟩
    |   ⟨executable_action⟩
    |   ⟨executable_rule⟩
    |   ⟨expression⟩
    |   ⟨ternary_operation⟩
expression::= ⟨expression_bracket⟩
    |   [ ⟨expression_logical_and⟩ [ [ ⟨OR⟩ ⟨expression⟩ ] ]* ]
expression_bracket::= [ ⟨LEFTROUNDBRACKET⟩ ⟨expression⟩ ⟨RIGHTROUNDBRACKET⟩
        ]
expression_logical_and::= [ ⟨expression_logical_xor⟩ [ [ ⟨AND⟩ ⟨expression⟩
        ] ]* ]
expression_logical_element::= ⟨logicalvalue⟩
    |   ⟨variable⟩
    |   ⟨executable_action⟩
    |   ⟨executable_rule⟩
    |   ⟨unification⟩
expression_logical_negation::= [ ⟨STRONGNEGATION⟩ ⟨expression⟩ ]
expression_logical_xor::= [ [ ⟨expression_logical_negation⟩
    |   ⟨expression_logical_element⟩
    |   ⟨expression_numeric⟩ ] [ [ ⟨XOR⟩ ⟨expression⟩ ] ]* ]
```

expression_numeric::= [⟨expression_numeric_relation⟩ [[[⟨EQUAL⟩
| ⟨NOTEQUAL⟩] ⟨expression_numeric⟩]]?]

expression_numeric_additive::= [⟨expression_numeric_multiplicative⟩ [[[
⟨PLUS⟩
| ⟨MINUS⟩] ⟨expression_numeric⟩]]?]

expression_numeric_element::= ⟨number⟩
| ⟨variable⟩
| ⟨executable_action⟩
| ⟨executable_rule⟩
| [⟨LEFTROUNDBRACKET⟩ ⟨expression_numeric⟩ ⟨RIGHTROUNDBRACKET⟩
]

expression_numeric_multiplicative::= [⟨expression_numeric_power⟩ [[[⟨SLASH⟩
| ⟨MODULO⟩
| ⟨MULTIPLY⟩] ⟨expression_numeric⟩]]?]

expression_numeric_power::= [⟨expression_numeric_element⟩ [[⟨POW⟩ ⟨expression_numeric⟩
]]?]

expression_numeric_relation::= [⟨expression_numeric_additive⟩ [[[⟨LESS⟩
| ⟨LESSEQUAL⟩
| ⟨GREATER⟩
| ⟨GREATEREQUAL⟩] ⟨expression_numeric⟩]]?]

floatnumber::= [⟨MINUS⟩? [[⟨DIGIT⟩+ ⟨DOT⟩ ⟨DIGIT⟩+]
| ⟨constant⟩]]

integernumber::= ⟨integernumber_positive⟩
| ⟨integernumber_negative⟩

integernumber_negative::= [⟨MINUS⟩ ⟨DIGIT⟩+]

integernumber_positive::= [⟨PLUS⟩? ⟨DIGIT⟩+]

literal::= [[⟨AT⟩
| ⟨STRONGNEGATION⟩]? ⟨atom⟩ [[⟨LEFTROUNDBRACKET⟩ ⟨termlist⟩?
⟨RIGHTROUNDBRACKET⟩]]? [[⟨LEFTANGULARBRACKET⟩ ⟨literalset⟩?
⟨RIGHTANGULARBRACKET⟩]]?]

literalset::= [⟨literal⟩ [[⟨COMMA⟩ ⟨literal⟩]]*]

logicalvalue::= ⟨TRUE⟩
| ⟨FALSE⟩

number::= ⟨floatnumber⟩
| ⟨integernumber⟩

string::= ⟨SINGLEQUOTESTRING⟩
| ⟨DOUBLEQUOTESTRING⟩

```
term::= ⟨string⟩
   |  ⟨number⟩
   |  ⟨logicalvalue⟩
   |  ⟨literal⟩
   |  ⟨variable⟩
   |  ⟨variablelist⟩
   |  [ ⟨LEFTANGULARBRACKET⟩ ⟨termlist⟩ ⟨RIGHTANGULARBRACKET⟩ ]
   |  ⟨expression⟩
   |  ⟨ternary_operation⟩

termlist::= [ ⟨term⟩ [ [ ⟨COMMA⟩ ⟨term⟩ ] ]* ]

ternary_operation::= [ ⟨expression⟩ ⟨ternary_operation_true⟩ ⟨ternary_operation_false⟩
       ]

ternary_operation_false::= [ ⟨COLON⟩ ⟨executable_term⟩ ]

ternary_operation_true::= [ ⟨QUESTIONMARK⟩ ⟨executable_term⟩ ]

unaryoperator::= ⟨INCREMENT⟩
   |  ⟨DECREMENT⟩

unification::= [ ⟨AT⟩? ⟨RIGHTSHIFT⟩ [ ⟨literal⟩
   |  [ ⟨LEFTROUNDBRACKET⟩ ⟨literal⟩ ⟨COMMA⟩ ⟨unification_constraint⟩
       ⟨RIGHTROUNDBRACKET⟩ ] ] ]

unification_constraint::= ⟨variable⟩
   |  ⟨expression⟩

variable::= [ ⟨AT⟩? [ ⟨UPPERCASELETTER⟩
   |  ⟨UNDERSCORE⟩ ] [ ⟨LOWERCASELETTER⟩
   |  ⟨UPPERCASELETTER⟩
   |  ⟨UNDERSCORE⟩
   |  ⟨DIGIT⟩
   |  ⟨SLASH⟩ ]* ]

variable_evaluate::= [ ⟨variable⟩ [ [ ⟨LEFTROUNDBRACKET⟩ ⟨termlist⟩ ⟨RIGHTROUNDBRACKET⟩
       ] ]? ]

variablelist::= [ ⟨LEFTANGULARBRACKET⟩ ⟨variable⟩ [ [ ⟨VLINE⟩ ⟨variable⟩ ]
       ]* ⟨RIGHTANGULARBRACKET⟩ ]
```

A.5 Terminal Grammar

AND::= ‘&&’
ASSIGN::= ‘=’
ASSIGNDECREMENT::= ‘-=’
ASSIGNDIVIDE::= ‘/=’
ASSIGNINCREMENT::= ‘+=’
ASSIGNMULTIPLY::= ‘*=’
AT::= ‘@’
ATOMIC::= ‘atomic’
AVOGADRO::= ‘avogadro’
BLOCKCOMMENT::= [‘/*’ ‘any char’* ‘*/’]
BOLTZMANN::= ‘boltzmann’
COLON::= ‘:’
COMMA::= ‘,’
DECONSTRUCT::= ‘=..’
DECREMENT::= ‘–’
DIGIT::= ‘0-9’
DOLLAR::= ‘$’
DOT::= ‘.’
DOUBLEEXCLAMATIONMARK::= ‘!!’
DOUBLEQUOTESTRING::= [‘"’ (*not* ‘"’)* ‘"’]
ELECTRON::= ‘electron’
EQUAL::= ‘==’
EULER::= ‘euler’
EXCLAMATIONMARK::= ‘!’
FALSE::= ‘false’
| ‘fail’
GRAVITY::= ‘gravity’
GREATER::= ‘>’
GREATEREQUAL::= ‘>=’
INCREMENT::= ‘++’
INFINITY::= ‘infinity’
LEFTANGULARBRACKET::= ‘[’
LEFTARROW::= ‘<-’
LEFTCURVEDBRACKET::= ‘{’
LEFTROUNDBRACKET::= ‘(’
LEFTSHIFT::= ‘«’
LESS::= ‘<’
LESSEQUAL::= ‘<=’
LIGHTSPEED::= ‘lightspeed’
LINECOMMENT::= [‘//’
| ‘#’ ‘any char’* ‘\r’? ‘\n’]
LOWERCASELETTER::= ‘a-z’
MINUS::= ‘-’
MODULO::= ‘%’
MULTIPLY::= ‘*’
NEUTRON::= ‘neutron’
NOTEQUAL::= ‘\\==’
| ‘!=’
OR::= ‘||’
PARALLEL::= ‘parallel’
PI::= ‘pi’
PLUS::= ‘+’
POW::= ‘**’
PROTON::= ‘proton’
QUESTIONMARK::= ‘?’
RIGHTANGULARBRACKET::= ‘]’
RIGHTARROW::= ‘->’
RIGHTCURVEDBRACKET::= ‘}’
RIGHTROUNDBRACKET::= ‘)’
RIGHTSHIFT::= ‘»’

RULEOPERATOR::= ‘:-’

SCORE::= ‘score’

SEMICOLON::= ‘;’

SINGLEQUOTESTRING::= [‘\’’ (*not* ‘\’’)* ‘\’’]

SLASH::= ‘/’

STRONGNEGATION::= ‘~’

TRUE::= ‘true’
| ‘success’

UNDERSCORE::= ‘_’

UPPERCASELETTER::= ‘A-Z’

VLINE::= ‘|’

WHITESPACE::= ‘ ’
| ‘\n’
| ‘\t’
| ‘\r’+

XOR::= ‘^’

B Testing Agent

```
// --- initial beliefs -----

~hallo("text").
hallo(123)[abc(8),value('xxxx')].
hallo(666)[abc(8)].
hallo(123).
hallo("foo").
hallo(1111).
hallo(600).
hallo(999).
hallo(900).
hallo(888).
hallo(777).
hallo(700).
hallo(foo(3)).
foo(blub(1),hallo("test")).
second(true).



// --- initial goal -----

!main.


// --- logical rules -----

fibonacci(X, R)
  // order of the rules are indeterministic, so for avoid
      indeterministic behaviour
  // add the condition, when the rule can be executed first
  :- X <= 2;  R = 1
  :- X > 2;   TA = X - 1; TB = X - 2; $fibonacci(TA,A); $fibonacci(TB,B)
      ; R = A+B
.

ackermann(N, M, R)
  :- N == 0; M > 0; R = M+1
  :- M == 0; N > 0; TN = N-1; $ackermann(TN, 1, RA); R = RA
  :- N > 0; M > 0; TN = N-1; TM = M-1; $ackermann(N, TM, RI); $ackermann
      (TN, RI, RO); R = RO
.

myfunction(X) :- generic/print("my logical rule", X).



// --- plans -----

+counter(X) <- generic/print("belief 'counter' added with variable value
     [", X, "] in Cycle [", Cycle, "]").
```

```
50 +beliefadd(X) <- generic/print("adds the 'beliefadd' with value [", X, "
       ] in Cycle [", Cycle, "]"); -beliefadd(X).
51 -beliefadd(X) <- generic/print("removes the 'beliefadd' with value [", X
       , "] in Cycle [", Cycle, "]").


54 +!mytest <- generic/print("my test plan without variable in cycle [",
       Cycle, "]").


57 +!mytest(X) <-
58   generic/print("my test plan with variable value [", X, "] in cycle[",
         Cycle, "]");
59   Y = X-1;
60   !mytest(Y)
61 .


64 -!errorplan <- generic/print("fail plan (deletion goal) in cycle [",
       Cycle, "]").


67 +!errorplan <-
68   generic/print("fail plan is failing in cycle [", Cycle, "]");
69   fail
70 .


73 -!myexternal <- generic/print("external trigger in cycle [", Cycle, "]")
       .


76 +!main

78   : >>( hallo(X), generic/type/isstring(X) ) <-
79     generic/print("---", "first plan", "---", "unification variables", X
           )

81   : >>( hallo(X), generic/type/isnumeric(X) && X > 1000 )  <-
82     generic/print("---", "second plan", "---", "unification variables",
           X)

84   <-

86     // --- internal variables -----

88     generic/print("constants", Score, Cycle, "    ", MyConstInt,
           MyConstString, "    ", PlanFail, PlanFailRatio, PlanSuccessful,
           PlanSuccessfulRatio);



92     // --- collections -----
```

```
94      L = collection/list/range(1, 20);
95      [ A|B|C| _ |D|E|F|G ] = L;
96      Intersect = collection/list/intersect( [1,2,3,4,5], [3,4,5,6,7],
            [3,8,9,5] );
97      Union = collection/list/union( [1,2,3], [2,3,4], [3,4,5] );
98      SD = collection/list/symmetricdifference( [1,2,3], [3,4] );
99      CP = collection/list/complement( [1,2,3,4,5], [1,2] );

101     generic/print("list elements", A,B,C,D,E,F,G, L);
102     generic/print("intersection & union & symmetric difference &
            complement", Intersect, "--", Union, "--", SD, "--", CP);
103     generic/print();



107     // --- literal accessing -----

109     [O|P] =.. foo( blub(1), blah(3) );
110     [H|I] = P;
111     generic/print("deconstruct", O,P,H,I);
112     generic/print();



116     // --- simple arithmetic -----

118     Z = 10 * 4 ** 0.5;
119     generic/print("simple expression", Z);
120     generic/print();



124     // --- string -----

126     SBase64 = generic/string/base64encode( "Base64 encoded string" );
127     SReverse = generic/string/reverse( "abcdefg" );
128     SUpper = generic/string/upper("AbCdefg");
129     SLower = generic/string/lower("AbCdefg");
130     SReplace = generic/string/replace( "a1b1defg1xyz1ui", "1", "-" );
131     SRand = generic/string/random( 20, "
            abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
            );
132     generic/print("string", SBase64, "--", SReverse, "--", SUpper, "--",
             SLower, "--", SRand, "--", SReplace);
133     generic/print();



137     // --- blas arithmetic -----

139     M = math/blas/matrix/create(2,2);
140     math/blas/matrix/set(M, 0,0, 1);
141     math/blas/matrix/set(M, 0,1, 2);
142     math/blas/matrix/set(M, 1,0, 3);
143     math/blas/matrix/set(M, 1,1, 4);
```

```
144     Det = math/blas/matrix/determinant(M);
145     EV  = math/blas/matrix/eigenvalue(M);
146     generic/print("matrix", M,Det,EV);
147     generic/print();



151     // --- random -----

153     Distribution = math/statistic/createdistribution( "normal", 20, 100
            );
154     RV = math/statistic/randomsample( Distribution, 8 );
155     generic/print("random", RV);
156     generic/print();



160     // --- date / time -----

162     [Hour | Minute | Second | Nano | Zone1 ] = generic/datetime/time();
163     [Day | Month | Year | DayOfWeek | DayOfYear | Zone2 ] =
            generic/datetime/date();
164     generic/print("date & time", Hour, Minute, Second, Nano, Zone1, "--"
            , Day, Month, Year, DayOfWeek, DayOfYear, Zone2);
165     generic/print();



169     // --- math functions -----

171     MinIdx = math/minindex(RV);
172     MaxIdx = math/maxindex(RV);
173     InRect = math/shape/inrectangle( 2,1,  0,0, 4,5);
174     InCircle = math/shape/incircle( 2,1,  2,2, 1);
175     InTriangle = math/shape/intriangle( 160,270,  350,320,  25,375,
            40,55 );

177     generic/print("min & max index", MinIdx, MaxIdx);
178     generic/print("shapes (in)", "", "rectangle", InRect, "circle",
            InCircle, "triangle", InTriangle);
179     generic/print();



183     // --- statistics -----

185     Statistic = math/statistic/createstatistic();
186     math/statistic/addstatisticvalue(Statistic, RV, L);

188     SMax = math/statistic/getstatisticvalue(Statistic, "max");
189     SMin = math/statistic/getstatisticvalue(Statistic, "min");
190     SCount = math/statistic/getstatisticvalue(Statistic, "count");
191     SPopVariance = math/statistic/getstatisticvalue(Statistic, "
            populationvariance");
192     SQuadraticMean = math/statistic/getstatisticvalue(Statistic, "
```

```
        quadraticmean");
193     SSecondMom = math/statistic/getstatisticvalue(Statistic, "
        secondmoment");
194     SStd = math/statistic/getstatisticvalue(Statistic, "
        standarddeviation");
195     SSum = math/statistic/getstatisticvalue(Statistic, "sum");
196     SSumSq = math/statistic/getstatisticvalue(Statistic, "sumsquare");
197     SVar = math/statistic/getstatisticvalue(Statistic, "variance");
198     SMean = math/statistic/getstatisticvalue(Statistic, "mean");

200     generic/print("statistic", SMax, SMin, SCount, SPopVariance,
        SQuadraticMean, SSecondMom, SStd, SSum, SSumSq, SVar, SMean );
201     generic/print();


204     FValue = collection/list/create(1, 3, 1, 1);
205     Fitness1 = math/statistic/fitnessproportionateselection( ["a", "b",
        "c", "d"], FValue );
206     Fitness2 = math/statistic/fitnessproportionateselection( ["e", "f",
        "g", "h"], [1,1,3,1] );
207     generic/print( "fitness proportionate selection", Fitness1, Fitness2
         );
208     generic/print();



212     // --- LP solver -----

214     LP1 = math/linearprogram/create( 2, 2, 1, 0 );
215     math/linearprogram/valueconstraint( LP1, 1, 1, 0, ">=", 1 );
216     math/linearprogram/valueconstraint( LP1, 1, 0, 1, ">=", 1 );
217     math/linearprogram/valueconstraint( LP1, 0, 1, 0, ">=", 1 );
218     [LP1Value | LP1PointCount | LP1Points] = math/linearprogram/solve(
        LP1, "minimize", "non-negative" );

220     LP2 = math/linearprogram/create( 0.8, 0.2, 0.7, 0.3, 0.6, 0.4, 0 );
221     math/linearprogram/valueconstraint( LP2, 1, 0, 1, 0, 1, 0,  "=", 23
        );
222     math/linearprogram/valueconstraint( LP2, 0, 1, 0, 1, 0, 1,  "=", 23
        );
223     math/linearprogram/valueconstraint( LP2, 1, 0, 0, 0, 0, 0, ">=", 10
        );
224     math/linearprogram/valueconstraint( LP2, 0, 0, 1, 0, 0, 0, ">=", 8 )
        ;
225     math/linearprogram/valueconstraint( LP2, 0, 0, 0, 0, 1, 0, ">=", 5 )
        ;
226     [LP2Value | LP2PointCount | LP2Points] = math/linearprogram/solve(
        LP2, "maximize", "non-negative" );

228     generic/print("LP solve minimize", LP1Value, LP1PointCount,
        LP1Points);
229     generic/print("LP solve maximize", LP2Value, LP2PointCount,
        LP2Points);
230     generic/print();
```

```
234     // --- polynomial interpolation -----
236     PI = math/interpolate/create("neville", [-5,1,2,8,14], [7,3,7,4,8]);
237     [PIV] = math/interpolate/interpolate( PI, 3 , 5, 10, -3);
239     generic/print("interpolate", PIV);
240     generic/print();
244     // --- hash -----
246     HashMD5 = crypto/hash( "md5", "hallo" );
247     HashMurmur = crypto/hash( "murmur3-32", "hallo" );
248     HashAdler = crypto/hash( "adler-32", "hallo" );
249     HashCrc = crypto/hash( "crc-32", "hallo" );
250     HashSHA = crypto/hash( "sha-256", "string test1", "second data", 4,
            5, 6);
251     generic/print("MD5 & SHA-256 & Murmur & Adler & CRC hash", HashMD5,
            HashSHA, HashMurmur, HashAdler, HashCrc);
252     generic/print();
256     // ---- crypto (AES & DES) -----
258     DESKey = crypto/createkey( "DES" );
259     DESEncrypt = crypto/encrypt( DESKey, "DES uncrypted message");
260     DESDecrypt = crypto/decrypt( DESKey, DESEncrypt);
261     generic/print( "crypto des", DESEncrypt, DESDecrypt );
263     AESKey = crypto/createkey( "AES" );
264     AESEncrypt = crypto/encrypt( AESKey, "AES uncrypted message");
265     AESDecrypt = crypto/decrypt( AESKey, AESEncrypt);
266     generic/print( "crypto aes", AESEncrypt, AESDecrypt );
267     generic/print();
271     // --- crypto (RSA) -----
273     [ PublicKey1 | PrivateKey1 ] = crypto/createkey( "RSA" );
274     [ PublicKey2 | PrivateKey2 ] = crypto/createkey( "RSA" );
276     Encrypt1to2 = crypto/encrypt( PublicKey2, "RSA message from 1 to 2"
            );
277     Encrypt2to1 = crypto/encrypt( PublicKey1, "RSA message from 2 to 1"
            );
279     Decrypt1to2 = crypto/decrypt( PrivateKey2, Encrypt1to2 );
280     Decrypt2to1 = crypto/decrypt( PrivateKey1, Encrypt2to1 );
282     generic/print("crypto rsa 1 to 2", Encrypt1to2, Decrypt1to2 );
```

```
283     generic/print("crypto rsa 2 to 1", Encrypt2to1, Decrypt2to1 );
284     generic/print();




289     // ---- sequential & parallel lambda expression -----

291     (L) -> Y : generic/print(Y);

293     BL = generic/agent/belieflist( "hallo" );
294     (BL) -> Y : generic/print(Y);

296     @(L) -> Y | R : R = Y+1;
297     generic/print("lambda return", R);
298     generic/print();

300     PL = generic/agent/planlist();
301     PLN = collection/map/keys(PL);
302     (PLN) -> Y : generic/print(Y);

304     generic/print();



308     // --- sequential & parallel unification -----

310     // unify default
311     >>hallo( UN1 ) << true;
312     >>foo( UN4, UN5 ) << true;
313     >>foo( blub( UN6 ), hallo( UN7 ) ) << true;
314     >>foo( blub(1), hallo( UN8 ) ) << true;

316     // unify with expression
317     >>( hallo( UN2 ), generic/type/isstring(UN2) ) << true;
318     @>>( hallo( UN3 ), generic/type/isnumeric(UN3) && (UN3 > 200) ) <<
            true;

320     // unify variable (I is defined on the deconstruct call on top)
321     >>( blah(UN9), I ) << true;

323     // manual literal parsing & unification
324     UN10 = generic/type/parseliteral("foo(12345)");
325     >>( foo(UN11), UN10 ) << true;

327     generic/print("unification", UN1, UN2, UN3, "   ", UN4, UN5, "   ",
            UN6, UN7, UN8, "   ", UN9, "   ", UN10, UN11 );
328     generic/print();



332     // --- repair & plan & goal handling -----

334     // test-goal not exist, so use repair definition
335     ?plannotexist << true;
```

```
337     // test-goal exists, so no repair handling (only the functor is
            checked)
338     ?main;
339
340     // run plan immediately
341     PLAN = "mytest";
342     !!PLAN;
343     !!PLAN(5);
344
345     // run plan within the next cycle
346     !mytest;
347     !mytest(4);
348     !errorplan;
349
350
351
352     // --- test belief calls -----
353
354     +beliefadd(UN8);
355
356
357
358     // --- rule execution -----
359
360     $myfunction("fooooooo");
361     $fibonacci(8, FIB);
362
363     RULE = "fibonacci";
364     $RULE(8,FIB2);
365
366     generic/print("rule execution (fibonacci)", FIB, FIB2);
367     FIB == 21;
368     FIB2 == 21;
369
370     $ackermann(3, 3, ACK);
371     generic/print("rule execution (ackermann)", ACK);
372     ACK == 61;
373
374     FIBX = -1;
375     $fibonacci(8, FIBX);
376     generic/print("rule execution (fibonacci) in-place modification",
            FIBX );
377     FIBX == 21;
378
379
380
381     // --- condition & plan passing -----
382
383     Text = Z > 100 ? "Z greater equal 100" : "Z is less 100";
384     generic/print("ternary operator", Text);
385
386     Z < 100;
387     generic/print("plan passed")
388 .
```

References

[1] Y. Shoham, "Agent-Oriented Programming," *Artificial Intelligence*, vol. 60, no. 1, pp. 51–92, 1993.

[2] A. S. Rao and M. P. Georgeff, "BDI agents: From theory to practice," in *Proc. 1st Int. Conf. on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pp. 312–319, 1995.

[3] A. S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *Proc. of MAAMAW '96*, (Secaucus, NJ, USA), pp. 42–55, Springer-Verlag New York, Inc., 1996.

[4] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. Wiley & Sons, 2007.

[5] TIOBE. `http://www.tiobe.com/tiobe_index`, accessed: 2016-06-27 (archived by WebCite® at `http://www.webcitation.org/6iZwpVq0y`).

[6] RedMonk. `http://redmonk.com/sogrady/2016/02/19/language-rankings`, accessed: 2016-06-27 (archived by WebCite® at `http://www.webcitation.org/6iZxPEb9K`).

[7] PopularitY. `http://pypl.github.io/`, accessed: 2016-06-27 (archived by WebCite® at `http://www.webcitation.org/6iZxjsbBs`).

[8] J. P. Müller and K. Fischer, "Application impact of multi-agent systems and technologies: A survey," in *Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages, and Frameworks* (O. Shehory and A. Sturm, eds.), pp. 27–53, Springer, 2014.

[9] T. Ahlbrecht, J. Dix, M. Köster, P. Kraus, and J. P. Müller, "An architecture for scalable simulation of systems of cognitive agents," *International Journal of Agent-Oriented Software Engineering*, 2016. To appear.

[10] M. Georgeff and A. Lansky, "Procedural knowledge," *Proceedings of the IEEE*, vol. 74, no. 10, pp. 1383–1398, 1986.

[11] M. P. Georgeff and A. L. Lansky, "Reactive reasoning and planning," in *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 2*, AAAI'87, pp. 677–682, AAAI Press, 1987.

[12] M. d'Inverno, M. Luck, M. Georgeff, D. Kinny, and M. Wooldridge, "The dMARS architecture: A specification of the distributed multi-agent reasoning system," *Autonomous Agents and Multi-Agent Systems*, vol. 9, pp. 5–53, jul 2004.

[13] V. Mascardi, D. Demergasso, and D. Ancona, "Languages for programming BDI-style agents: an overview.," in *WOA*, pp. 9–15, 2005.

[14] R. H. Bordini, A. L. Bazzan, R. de O Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser, "AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling," in *Proc. 1st Int. Joint Conf. on Autonomous Agents and Multiagent Systems: part 3*, pp. 1294–1302, ACM, 2002.

[15] S. Dennisen and J. P. Müller, "Iterative committee elections for collective decision-making in a ride-sharing application," in *Proc. 9th International Workshop on Agents in Traffic and Transport (ATT 2016) at IJCAI 2016* (A. L. C. Bazzan, F. Klügl, S. Ossowski, and G. Vizzari, eds.), (New York, USA), pp. 1–8, CEUR, July 2016. Electronic proceedings.

[16] A. Zatelli, Maicon R.and Ricci and J. F. Hübner, "A concurrent architecture for agent reasoning cycle execution in jason," in *Multi-Agent Systems and Agreement Technologies: 13th Europ. Conf., EUMAS 2015, and 3rd Int. Conf., AT 2015, Athens, Greece, 2015*, pp. 425–440, Springer International Publishing, 2016.

[17] M. J. Wooldridge, "An introduction to multiagent systems," 2009.

[18] A. Tanenbaum and H. Bos, *Modern Operating Systems: Global Edition*. Pearson Education Limited, 2015.

[19] A. Lux and D. Steiner, "Understanding cooperation: An agent's perspective.," in *ICMAS*, pp. 261–268, 1995.

[20] J. P. Müller, *The Design of Intelligent Agents*, vol. 1177 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996.

[21] S. Rabin and N. Sturtevant, "Combining bounding boxes and jps to prune grid pathfinding," *AAAI Conference on Artificial Intelligence*, 2016.